This assignment is due by the due date on the dropbox.

This exercise is the second part of a multipart assignment to build a small MVC-based login system with some basic per-user functionality.

In this section, you will be building out the functionality for each user to make their own characters and attach data to them. The users will also be able to retrieve only their characters.

You will be starting where you left off in DomoMaker-A. Continue working in the same repo and Heroku app.

1. First, we need to add sessions so that we can accurately track who logged in and who they are. In a stateless transaction, every transaction is new, so we need a ticket to identify this person. We use sessions to handle that for us.

    A session is a uniquely generated key that is tracked by the server to map to an individual user. The unique key is then sent to the user as a cookie that the server can track. In each request, the cookies are sent back to the server, in which it will then see if it has a session id that matches that unique cookie.

    Cookies are just key:value pairs set by the server or browser for tracking purposes.
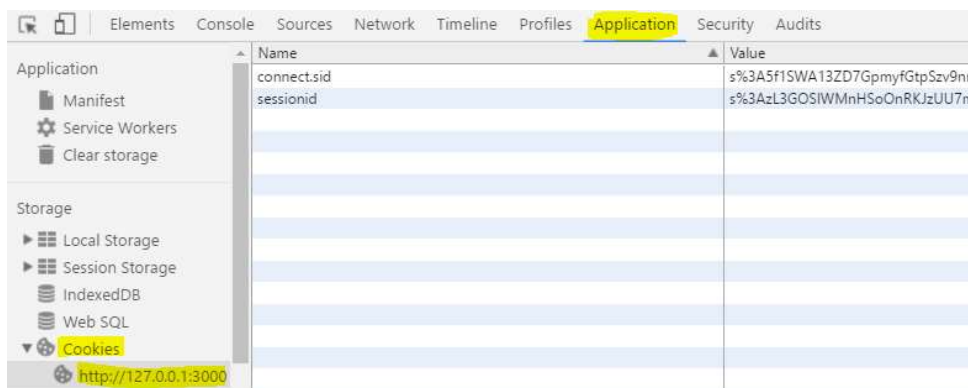
    As such, you want to make sure you secure your cookies and refresh them over time. This usually means automatically logging users out or just sending them a new session id back sometimes. This limits how long a key is available, and how much time in which it could potentially be stolen. There are added protections against this that we will also add later.

2. In your server/app.js pull in the express-session library.

```
const path = require('path');
const express = require('express');
const compression = require('compression');
const favicon = require('serve-favicon');
const cookieParser = require('cookie-parser');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');
const expressHandlebars = require('express-handlebars');
const helmet = require('helmet');
const session = require('express-session');
```

3. In your app.js, add a section for session configuration. To do this, tell your express app to use express-session. The express-session module takes several configuration options as a JSON object. Once it is configured, the session module will add a session object to every request object which can be accessed by saying "req.session". This session object can be used to track and store information unique to that user between requests.

```
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

app.use(session({
    key: 'sessionid',
    secret: 'Domo Arigato',
    resave: false,
    saveUninitialized: false,
}));

app.engine('handlebars', expressHandlebars.engine({ defaultLayout: '' }));
```

The key is the name of your cookie so that it can be tracked in requests. When the browser sends requests the cookies will come as well. The key setup in the session options will tell our session module which cookie to look for when looking for a session cookie.

The default session key name is "connect.sid" which tells the clients you are using the connect module (what express is based on). This is considered a mild security flaw because it indicates to people what modules/frameworks they should be looking

for security holes in. Instead you rename it to something less telling about the framework, such as "sessionid" or anything you want.

The secret is a private string used as a seed for hashing/creating unique session keys. This makes it so your unique session keys are different from other servers using express. The secret can be changed to anything you want, but will invalidate existing session ids (which isn't necessarily a huge issue).

The resave option set to false tells the session library to only send the session key back to the database if it changes. If it were set to true, we would generate a lot of database requests that were unnecessary.

The saveUninitialized option set to false prevents us from saving uninitialized sessionids to the database.

4. Now we will add some basic session functionality to our main controllers. Open the Account.js Controller. Add the following line to destroy the session. With the express-session module, every request will have a session object in it that manages the user's session and session variables.

```
const logout = (req, res) => {
  req.session.destroy();
  res.redirect('/');
}
```

The destroy function will remove a user's session. We call this on logout so that our server knows they are no longer logged in.

5. Now we will store some basic data about our user in the session.

As mentioned previously, the req.session object also allows you to store data in the session about the user. You will not want to store a ton of data in the session object because it takes up memory and is not saved. You can use it to store a decent number of variables though.

This means once a user is logged in and has a session, you can store information about them that you don't want to constantly look up in the database. Eventually this data will be backed up in redis, and will not be creating state on our server.

In the login function of the Account controller, add a new variable to req.session called account. We will set this equal to the toAPI call from our model.

Look at the toAPI function in the Account.js Model to see what it attaches. When a user logs in, we will attach all of the fields from toAPI to their session for tracking.

```
return Account.authenticate(username, pass, (err, account) => {
  if (err || !account) {
    return res.status(401).json({ error: 'Wrong username or password!' });
  }

  req.session.account = Account.toAPI(account);

  return res.json({ redirect: '/maker' });
});
```

6. Similarly, **add a line to the signup function in the Account controller** to attach the account data from toAPI. Since the user is signing up and being logged in automatically, we need to duplicate the account data in the session just like they had logged in.

```
try {
  const hash = await Account.generateHash(pass);
  const newAccount = new Account({username, password: hash});
  await newAccount.save();
  req.session.account = Account.toAPI(newAccount);
  return res.json({ redirect: '/maker' });
} catch (err) {
```

7. Now start your node server and test it. Login or signup with an account. If you check out the application inspector in Chrome, you will see the cookie (under the name you gave in the express-session options) with a unique value.

*Note: You may still see "connect.sid" as a cookie since that is the default for Connect (which Express MVC is based on). This is because nothing ever told the browser to get rid of it. It's not used any more, but the browser will hang onto the cookie for a while unless the server tells it to delete the cookie immediately.
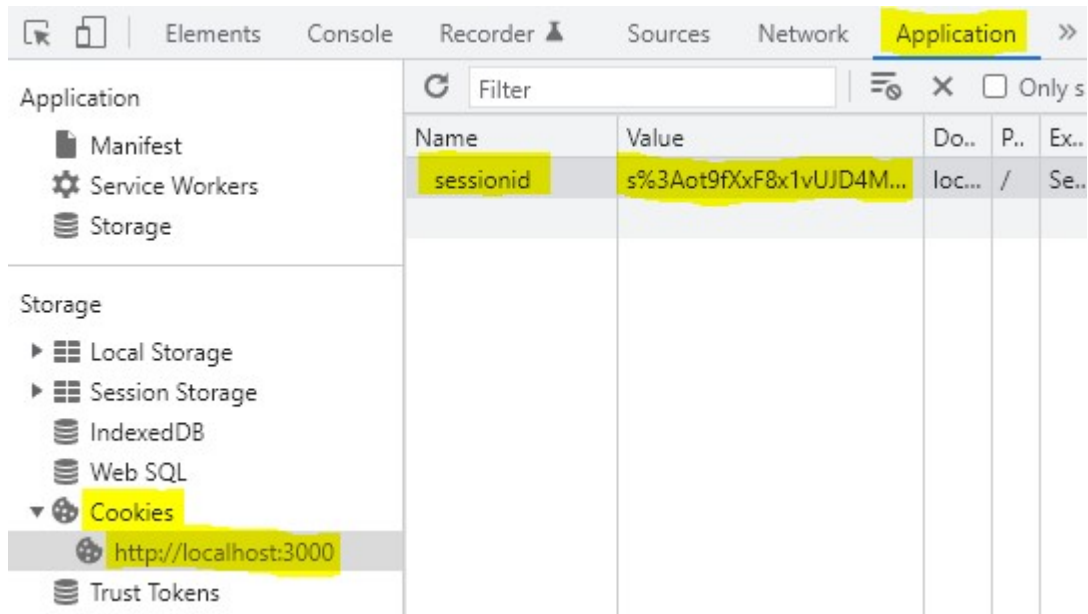
8. Logout of the website and login again. You should see the sessionid value change because your logout function *destroys* the session (thus making them a new user). Each time you login, a new key is generated for you.

> sessionid   s%3AIh8nZiXr8FnWyaejX...

9. Lets add more functionality to the Domo page. Create a new file in the Models folder called Domo.js

10. Inside of the index.js file in the Models folder, add an entry for the new Domo file.

```
module.exports.Account = require('./Account.js');
module.exports.Domo = require('./Domo.js');
```

11. Inside of the Domo.js Model, add the following

Part 1

```javascript
const mongoose = require('mongoose');
const _ = require('underscore');

const setName = (name) => _.escape(name).trim();

const DomoSchema = new mongoose.Schema({
    name: {
        type: String,
        required: true,
        trim: true,
        set: setName,
    },
    age: {
        type: Number,
        min: 0,
        required: true,
    },
    owner: {
        type: mongoose.Schema.ObjectId,
        required: true,
        ref: 'Account',
    },
    createdDate: {
        type: Date,
        default: Date.now,
    },
});
```

Part 2 of the Domo.js Model

```javascript
DomoSchema.statics.toAPI = (doc) => ({
    name: doc.name,
    age: doc.age,
});

const DomoModel = mongoose.model('Domo', DomoSchema);
module.exports = DomoModel;
```

12. Now open your client.js file inside of the client folder (where your client code is). Remember since this is a client side JS file, it cannot communicate with the server unless it is through fetch() or a similar system.

    In this file, notice that there is an event listener for when the domoForm submit button is pressed. Similar to our signup/login code, pressing that button will prevent the default browser action and create a POST request to the domoForm's ACTION attribute (inside app.handlebars). This fetch() call expects a response in JSON as specified in the Content-Type header set in sendPost().

13. Now open the app.handlebars view. Notice that the action goes to "/maker" with a POST method. This is where our form will submit to.

14. First, we need to create a POST route for /maker in the router. Add this "/maker" POST line to the router.js. It will call the Domo Controller's make function (which we have not made yet).

```javascript
const router = (app) => {
  app.get('/login', controllers.Account.loginPage);
  app.post('/login', controllers.Account.login);

  app.get('/signup', controllers.Account.signupPage);
  app.post('/signup', controllers.Account.signup);

  app.get('/logout', controllers.Account.logout);

  app.get('/maker', controllers.Domo.makerPage);
  app.post('/maker', controllers.Domo.makeDomo);

  app.get('/', controllers.Account.loginPage);
};
```

15. Open the Domo controller and pull in the Domo Model module at the top of the file.

```
const models = require('../models');
const Domo = models.Domo;
```

16. Next, we will make a makeDomo function and export it. This is very similar to our signup function in the Account controller. We need to create a new JSON object of our Domo data and pass it to the Domo Model. Once we have the database object, we just have to save it and handle any successes or errors.

Notice for the Domo owner field, we put the ID of the owner we stored in req.session from the Account Model's toAPI method. This is the nice part of storing a user's data in their session. We can access the session anywhere we can access the request.

Notice that the success returns a JSON object with a redirect field. This field is not automatic by any means. This is nearly identical to our signup/login JSON. Our client fetch() function is waiting for a redirect variable to come back from the server to know what page to load. This is all setup and controlled by us.

```
const makeDomo = async (req, res) => {
    if(!req.body.name || !req.body.age) {
        return res.status(400).json({ error: 'Both name and age are required!' });
    }

    const domoData = {
        name: req.body.name,
        age: req.body.age,
        owner: req.session.account._id,
    };

    try{
        const newDomo = new Domo(domoData);
        newDomo.save();
        return res.json({ redirect: '/maker' });
    } catch (err) {
        console.log(err);
        if(err.code === 11000) {
            return res.status(400).json({ error: 'Domo already exists!' });
        }
        return res.status(500).json({ error: 'An error occured making domo!' });
    }
}

module.exports = {
    makerPage,
    makeDomo,
}
```

17. Start your server, login and test adding a Domo. Note that if you (or nodemon) restart your server, you will need to log out and log back in to regenerate your session otherwise things will break. We will fix this with redis in DomoMaker C.

- Test the page with invalid data (should get an error message from the Domo)
- Test the page without all the fields (should get an error message from the Domo)
- Test the page with correct data (nothing happens/page reloads)

When we test the page with correct data, the page just reloads. Why? This is because that's all we do on success. We just sent JSON back with a URL to load (which is the same page it's already on, thus reloading the page). Not very exciting…Let's do more.

18. The next thing to do is to make sure we can actually see the Domos each user makes. Change the makerPage function to grab all of the Domos for a particular user (based on their user ID which we stored in their session ☺) and pass it to the view.

In the Domo controller, update the makerPage function with the following.

```
const makerPage = async (req, res) => {
    try{
        const query = {owner: req.session.account._id};
        const docs = await Domo.find(query).select('name age').lean().exec();

        return res.render('app', {domos: docs});
    } catch (err){
        console.log(err);
        return res.status(500).json({error: 'Error retrieving domos!'});
    }
};
```

19. Open the app.handlebars view. We need to add to our domo div tag with the following. Handlebars can iterate through variables passed in from the controller. Our makerPage controller passes in a list of each of the domos for a user that we can iterate over.

```
<section id="domos">
  {{#if domos}}
    {{#each domos}}
      <div class="domo">
        <img src="/assets/img/domoface.jpeg" alt="domo face" class="domoFace">
        <h3 class="domoName">Name: {{this.name}}</h3>
        <h3 class="domoAge">Age: {{this.age}}</h3>
      </div>
    {{/each}}
  {{else}}
    <h3 class="emptyDomo">No Domos yet!</h3>
  {{/if}}
</section>
```

20. Restart your server and reload the "/maker" page. You will likely get an error.

```
TypeError: Cannot read properties of undefined (reading '_id')
    at makerPage (C:\Users\Austin\Downloads\New folder\DomoMake
    at Layer.handle [as handle_request] (C:\Users\Austin\Downlo
    at next (C:\Users\Austin\Downloads\New folder\DomoMaker-A-S
```

This is because our sessions (which has our user ID and such) are stored in memory on the server. When the server restarts, we will lose all of that data. Second, we try to go to a page that requires the data, so it fails with an error.

In the next assignment, we will fix it so that our sessions are stored externally from node. We will also ensure people cannot get to pages that require a session without having a session in memory.

21. Go back to the login page. Log in and you should see the Domo you made before (now that the views are showing it).

Try making another Domo -

- Test the page with invalid data (should get an error message from the Domo)
- Test the page without all the fields (should get an error message from the Domo)
- Test the page with correct data (page reloads with update)

Now our new Domos are showing up when the page reloads. Since on success we reload the page, the makerPage controller gets called again, gathering all of the Domos for this user and putting them into a view.

You technically don't have to reload the page. You could actually just send back an updated JSON list to update the screen. A JSON response of the new Domo would actually be faster for the client & server than reloading the page constantly. You would just have to have the client JS listen for a JSON response from the server filled with data you could put on the screen. We won't do that in this assignment though. For now, we will just reload the page as we are doing.

22. Logout and log back in. Make sure the list stays the same. Fix any errors before moving on (database errors in the console are fine as long as it works. Those errors are probably just our console statements printing the error message when you try to input bad data).

23. Log back out and create a second account. Make a Domo on that second account. Ensure that works correctly.

24. Now logout and log back in with the second account. Make sure this works correctly and only shows the second user's Domos.

25. Last test – log out, log back in with the first account, ensure it works. Then log out and log back in with the second account. Check that it works correctly.

Now you can actually start tracking data and having users add data. If you were building a native client app like a native iOS app or native Android app, you wouldn't have to return a 'view' since the native apps already contain the whole compiled client. Instead this same server architecture would work, but you would always send JSON back, never render. Every request from the native app would come as an fetch()/REST request. We will start doing this later when we get to React in DomoMaker D.

26. Upload and test your app on Heroku to make sure it works correctly.

In the next assignment, we will tighten up the app considerably.

DOMOMAKER - B

ESLint & Code Quality

You are required to use ESLint with the AirBnB configuration for code quality checks. The eslintrc config file was given to you in the starter files. Your **server** code must pass ESLint to get credit for this part. Warnings are okay, but errors must be fixed. I won't require ESLint on the client side code in this assignment. ***If you cannot fix an error or need help, please ask us!***

You are also required to use the "pretest" script in order to run ESLint. I should be able to run "*npm test*" and have ESLint scan your code for me.

Continuous Integration

You are required to use CircleCI for continuous integration. Your build must succeed and be handed in with the assignment. The circle.yml config file was given to you in the starter files. CircleCI will build based on your *npm test* and ESLint configuration.

For instructions on how to do this, look at the ***CircleCI Setup*** instructions on mycourses.

Build & Bundling Scripts

Your app should have custom scripts to automatically restart node upon server changes (I suggest using nodemon) and create a client-side JS bundle (Webpack is installed and configured to do this with the "npm run webpack" command).

You should write your JS in separate files and create a bundled JS file from them. The only JS file that should go to the client is the bundled file.

-- Rubric on Next Page –

DOMOMAKER - B

Rubric

DESCRIPTION		SCORE	VALUE %
	App works correctly – all pages, forms and functions work correctly. App.js starts and runs correctly.		25
	Sessions connected and work correctly – Sessions are working correctly. When a user loads the page, they get a sessionid cookie.		15
	Logging out destroys session – When a user logs out, their session is destroyed and a new session is created (new sessionid cookie).		10
	Domo Model works as instructed – The Domo model works correctly and stores objects in the database.		15
	Can add Domos successfully – User can add a new Domo successfully. User is prevented from adding Domos with invalid data.		20
	App page retrieves Domos successfully – When the app page is loaded, a user is shown their Domos from the database. If they add a new Domo, this page is updated. If there are no Domos, then a message is displayed.		15
	Git Penality – If your code is not in a Git repo or the link is not handed in, there will be a penalty.		-30% (this time)
	Heroku Penalty – If your app is not functional on Heroku, there will be a penalty. This means you need to correctly hookup the addons on Heroku.		-30% (this time)
	CircleCI Penalty – If your build is not successful on CircleCI, there will be a penalty.		-30% (this time)
	Build and Bundling Penalty - If your package.json does not include properly working automatic build scripts for Node (nodemon or other) and client-side bundling scripts (babel, webpack, browserify, rollup or other), there will be a penalty.		-20% (this time)
	ESLint Penalties – I should be able to run 'npm test' and have ESLint scan your code without errors. Warnings are acceptable. **You may not alter the rules in the .eslintrc file provided.** **Check your code with ESLint often during development!** **If you cannot fix an error, please ask us!**	1 Error 2 Errors 3 Errors 4 Errors 5+ Errors	-10% -15% -20% -25% -35%
	Additional Penalties – These are point deductions for run time errors, poorly written code or improper code. There are no set values for penalties. The more penalties you make, the more points you will lose.		
	TOTAL		100%

By the due date please submit a zip of your project to the dropbox. **Do not** include the node_modules folder in the zip or in your git repo.

In the submission comments include the following:
- *a link to your Heroku App (not the dashboard but the working web link from 'open app'),*
- *a link to your Github repo (if private, please add the TA and myself).*
- a link to your circleCI build *in the submission comments.*